Estructuras de datos dinámicas:

¿Qué es una estructura de datos?

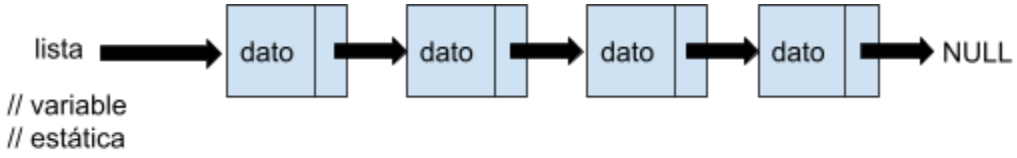
Se trata de un conjunto de variables de un determinado tipo agrupadas y organizadas de alguna manera para representar un comportamiento. Lo que se pretende con las estructuras de datos es facilitar un esquema lógico para manipular los datos en función del problema que haya que tratar y el algoritmo para resolverlo. En algunos casos la dificultad para resolver un problema radica en escoger la estructura de datos adecuada. Y, en general, la elección del algoritmo y de las estructuras de datos que manipulará estarán muy relacionadas.

Según su comportamiento durante la ejecución del programa distinguimos estructuras de datos:

- Estáticas: su tamaño en memoria es fijo. Ejemplo: arrays.
- Dinámicas: su tamaño en memoria es variable. Ejemplo: listas enlazadas con punteros, ficheros, etc.

Listas vinculadas

Conjunto de datos de acceso secuencial.
La lista vinculada es la estructura dinámica más simple, consiste de un conjunto de variables anónimas (creadas con malloc, y que no tienen nombre) que se vinculan unas a otras a través de un puntero.
Gráficamente, una lista (de cualquier tipo de dato) sería algo así:



A cada uno de los elementos de la lista vinculada le vamos a llamar NODO. El acceso al primer nodo de la lista se hace a través de una variable (con nombre) de tipo puntero. El final de la lista se determina con NULL. Cada nodo de la lista contiene campos con información y uno con la dirección de memoria del siguiente nodo. El acceso a cada nodo de la lista se hace en forma secuencial, a través del campo que contiene la dirección del siguiente, así cada nodo, nos proporciona la dirección donde se encuentra el que le sigue. La estructura de datos que se necesita para crear un nodo (y con ello la lista de nodos) es la siguiente:

```
typedef struct {
     tipo campo1;  // campos de datos, pueden ser estructuras
     tipo campo2;
     ........
      tipo campoN;
      struct nodo * siguiente; // campo con la dirección de memoria
del siguiente nodo
} nodo ;

nodo * lista ; // variable estática y con nombre que contiene la
// dirección de memoria del
// primer nodo de la lista. Si la lista está vacía, entonces
// lista = NULL;
```

Operaciones básicas del manejo de listas: Para poder manipular las listas en forma adecuada, lo haremos a través de un conjunto de funciones que se explicarán a continuación.

NOTA: Para ejemplificar la explicación, usamos la siguiente estructura de datos:

```
typedef struct {
     persona dato;
     struct nodo * siguiente;
} nodo;
```

```
typedef struct {
     char nombre[20];
     int edad;
} persona;
```

```
nodo * lista; // variable estática definida en el main()
```

donde:

***dato** es una variable de tipo persona que contiene información.*

***siguiente** es un campo puntero, que contiene la dirección de memoria de otra estructura similar.*

***lista** es una variable estática (la declaramos a nivel de main) que contiene la dirección de memoria del primer nodo de la lista.*

Inicializar la lista

Inicializa el puntero al primer nodo de la lista con el valor NULL.

```
nodo * inicLista() {
   return NULL;
}
```

Crear un nuevo nodo para luego agregar a la lista

Función que recibe como parámetro los campos de información para la estructura, los agrega a la misma y retorna un puntero al NODO creado.

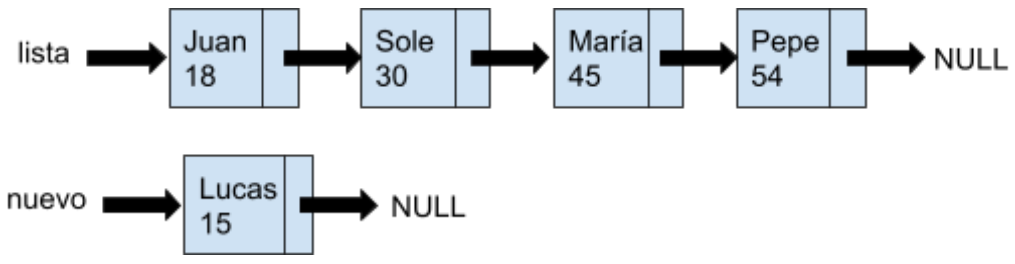
```
nodo * crearNodo (persona dato) {
     //crea un puntero de tipo nodo
     nodo * aux = (nodo*) malloc(sizeof(nodo));
     //asigna valores a sus campos de información
     aux->dato = dato;
     //asigna valor NULL al campo que contiene la dirección de memoria del
     //siguiente nodo
     aux->siguiente = NULL;
     //retorna la dirección de memoria del nuevo nodo, que deberá ser
     asignada a una variable de tipo "puntero a nodo".
     return aux;
}
```

Agregar un nodo ya creado al principio de la lista

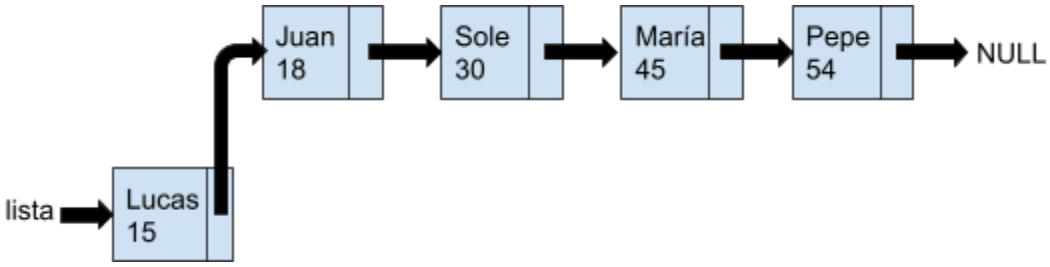
Función que recibe como parámetro una variable puntero al comienzo de la lista y otra variable puntero a un nuevo nodo. Agrega este nuevo nodo al comienzo de la lista y retorna la nueva posición de memoria.

```
 nodo * agregarPpio (nodo * lista, nodo * nuevoNodo) {
     //si la lista está vacía, ahora apuntará al nuevo nodo.
     if(lista == NULL) {
        lista = nuevoNodo;
     }else
     //si la lista no está vacía, inserta el nuevo nodo al comienzo de la
     //misma, y el viejo primer nodo pasa a ser el segundo de la lista.
     {
        nuevoNodo->siguiente = lista;
        lista = nuevoNodo;
     }
     return lista;
 }
```

Antes de la inserción:



Después de la inserción:



NOTA IMPORTANTE: debido al tipo de pasaje de parámetros (por valor), el contenido del parámetro lista (*un puntero al primer nodo*) no puede ser alterado una vez que la función ha terminado, o sea, el parámetro actual siempre conserva su valor. No confundirse, lo que sí se puede alterar es ***lista**.
Lo que hacíamos antes, era cambiar el contenido de una variable apuntada, y no el contenido de la variable apuntadora (que contiene una dirección de memoria).

Buscar el último nodo

Función que nos retorna la dirección de memoria del último nodo de la lista. Esta función solamente se invocará si la lista no está vacía (lista != NULL)

```
nodo * buscarUltimo(nodo * lista) {
   nodo * seg = lista;
   if(seg != NULL)
      while(seg->siguiente != NULL) {
         seg = seg->siguiente;
      }
    return seg;
 }
```

donde:

*la variable local **seg** contiene al principio, la dirección de memoria de primer nodo de la lista. Luego, al ir ejecutándose la instrucción **seg = seg->siguiente;** dentro del ciclo de repetición, **seg** va tomando la dirección del nodo siguiente. De esta forma se avanza en el recorrido secuencial de la lista.*

Utilizamos una variable local **seg** para recorrer la lista en vez del mismo parámetro **lista** para no alterar el contenido del mismo, aunque, debido al tipo de pasaje de parámetros (por valor) el contenido de **lista** no podría ser alterado nunca. Sí se puede alterar el contenido de ***lista.** Hacerlo de esta forma es una buena disciplina de programación que evita posibles errores.

Notar que en el ciclo de repetición, la condición es **(seg->siguiente != NULL)**, esto es porque debemos detener nuestro avance al llegar al último nodo, o sea, al nodo cuyo siguiente es NULL. Por este motivo evaluamos al siguiente y no al propio nodo.

Buscar un nodo según el valor de un campo

Función que retorna la dirección de memoria de un nodo que contiene el campo **dato.nombre** con el mismo valor que el parámetro **nombre**

```
nodo * buscarNodo(nodo * lista, char nombre[20]) {
   //busca un nodo por nombre y retorna su posición de memoria
   //si no lo encuentra retorna NULL.

   nodo * seg;  //apunta al nodo de la lista que está siendo procesado
   seg = lista; //con esto evito cambiar el valor de la variable
                //lista, que contiene un puntero al primer nodo de la
                //lista vinculada

   while ((seg != NULL) && ( strcmp(nombre, seg->dato.nombre)!=0 )) {
//busco mientras me quede lista por recorrer y no haya encontrado el nombre
      seg=seg->siguiente; //avanzo hacia el siguiente nodo.
   }
//en este punto puede haber fallado alguna de las dos condiciones
//del while. si falla la primera es debido a que no encontró lo
//que buscaba (seg es NULL), si falla la segunda es debido a que se
//encontró el nodo buscado.
   return seg;
}
```

Agregar un nuevo nodo al final de la lista

Esta función nos permite agregar al final de la lista un nuevo nodo.

```
nodo * agregarFinal(nodo * lista, nodo * nuevoNodo) {

   if(lista == NULL) {
      lista = nuevoNodo;
   } else {
      nodo * ultimo = buscarUltimo(lista);
      ultimo->siguiente = nuevoNodo;
   }
   return lista;
}
```

Notar que se invoca a la función **buscarUltimo(nodo *)** ya definida previamente.
La variable local **ultimo** contiene la dirección de memoria del último nodo de la lista, y a

través de la misma, accedo a su campo **siguiente**, actualizando su valor. Esto agrega un nodo más a la lista.
El retorno de la función siempre es el mismo, excepto la primera vez, cuando la lista está vacía y se le agrega el primer elemento.

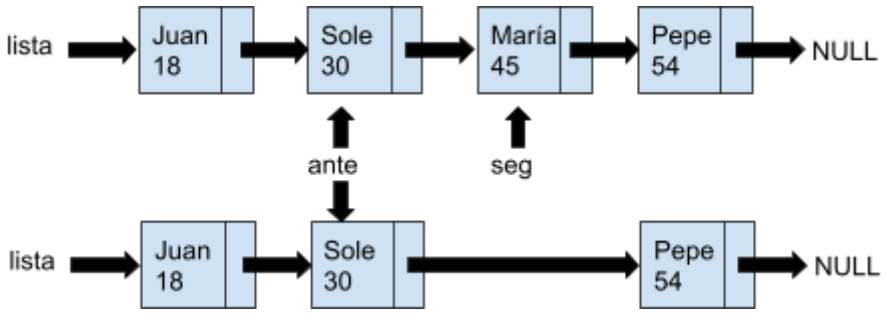
Borrar un nodo de la lista buscándolo por el valor de un campo

Función que nos permite borrar un nodo que está en el interior de la lista, buscándolo por el valor de uno de sus campos. Retorna un puntero al comienzo de la lista.
Generalmente el puntero al comienzo de la lista no se modifica, excepto cuando se elimina el primer nodo.

```
nodo * borrarNodo(nodo * lista, char nombre[20]) {
   nodo * seg;
   nodo * ante; //apunta al nodo anterior que seg.
   if((lista != NULL) && (strcmp(nombre, lista->dato.nombre)==0 )) {

      nodo * aux = lista;
      lista = lista->siguiente; //salteo el primer nodo.
      free(aux);                //elimino el primer nodo.
   }else {
      seg = lista;
      while((seg != NULL) && (strcmp(nombre, seg->dato.nombre)!=0 )) {
         ante = seg;           //adelanto una posición la variable ante.
         seg = seg->siguiente; //avanzo al siguiente nodo.
      }
      //en este punto tengo en la variable ante la dirección de
      //memoria del nodo anterior al buscado, y en la variable seg,
      //la dirección de memoria del nodo que quiero borrar.
      if(seg!=NULL) {
         ante->siguiente = seg->siguiente;
         //salteo el nodo que quiero eliminar.
         free(seg);
         //elimino el nodo.
      }
   }
   return lista;
}
```

ejemplo: eliminar el nodo “María”



Agregar un nodo nuevo manteniendo el orden (según un campo)

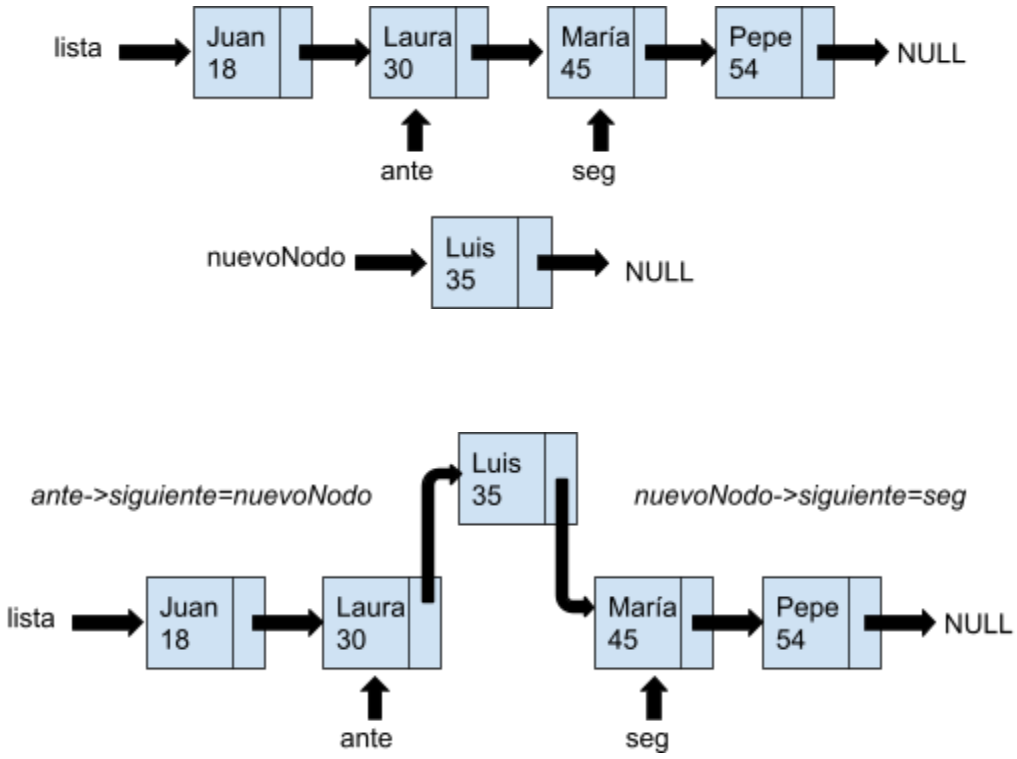
Esta función agrega un nodo nuevo a la lista, insertándolo en el lugar correspondiente según un orden preestablecido por un campo. Por ejemplo: insertar un nuevo nodo en la lista ordenada por el campo nombre. Retorna un puntero al primer nodo de la lista, que se modifica solamente cuando el nuevo nodo se inserta en la primera posición.

```
nodo * agregarEnOrden(nodo * lista, nodo * nuevoNodo) {
   // agrega un nuevo nodo a la lista manteniendo el orden.

   //si la lista está vacía agrego el primer elemento.
   if(lista == NULL) {
      lista = nuevoNodo;
   }else {
      //si el nuevo elemento es menor que el primero de la lista,
      //agrego al principio

      if(strcmp(nuevoNodo->dato.nombre,lista->dato.nombre)<0){
         lista = agregarPpio(lista, nuevoNodo);
      } else {
         //busco el lugar en donde insertar el nuevo elemento.
         //necesito mantener la dirección de memoria del nodo anterior
         //al nodo que tiene un nombre mayor al del nuevo nodo.
         nodo * ante = lista;
         nodo * seg = lista->siguiente;
         while((seg != NULL)
                  &&(strcmp(nuevoNodo->dato.nombre,seg->dato.nombre)>0)) {
            ante = seg;
            seg = seg->siguiente;
         }
         // inserto el nuevo nodo en el lugar indicado.
         nuevoNodo->siguiente = seg;
         ante->siguiente = nuevoNodo;
      }
   }
   return lista;
}
```

ejemplo: agregar el nodo “Luis”



Borrar toda la lista y liberar la memoria ocupada

Esta función borra toda la lista entera y libera todas las direcciones de memoria ocupada por sus nodos, retornando NULL.

```
nodo * borrarTodaLaLista(nodo * lista) {
   nodo * proximo;
   nodo * seg;
   seg = lista;
   while(seg != NULL) {
      proximo = seg->siguiente;  //tomo la dir del siguiente.
      free(seg);                 //borro el actual.
      seg = proximo;             //actualizo el actual con la dir del
                                 //siguiente, para avanzar.
   }
   return seg; // retorna NULL a la variable lista del main()
}
```

Sumar el contenido de cada nodo (el campo edad)

```
int sumarEdadesLista(nodo * lista) {
//recorro la lista y sumo las edades de los socios.
   int suma = 0;
   nodo * seg = lista;
   while (seg != NULL) {
      suma = suma + seg->dato.edad;
      seg = seg->siguiente;
   }
   return suma;
}
```

Eliminar el primer nodo de una lista

Queda como ejercicio para el alumno.

Eliminar el último nodo de una lista

Queda como ejercicio para el alumno.

Otras funciones

Para armar un sistema, usaremos también las siguientes funciones, que usan a las funciones anteriores.

```
void mostrarUnNodo(nodo * aux);
void recorrerYmostrar(nodo * lista);
nodo * subprogramaIngresarDatosAlFinal(nodo * lista);
nodo * subprogramaingresarDatosAlPpio(nodo * lista);
nodo * subprogramaAgregarUnNodoEnOrden(nodo * lista);
nodo * subprogramaCrearListaOrdenada(nodo * lista);
void subprogramaBusquedaDeUnNodo(nodo * lista);
nodo * subprogramaBorrarNodo(nodo * lista);
void menu();
```

Estas funciones son muy simples, por lo cual no se dará una explicación detallada. Se recomienda analizar el código de las mismas.